Web Security: Detection of Cross Site Scripting in PHP Web Application using Genetic Algorithm

Abdalla Wasef Marashdih[1], Zarul Fitri Zaaba[1]* & Herman Khalid Omer[2]
[1]School of Computer Sciences,Universiti Sains Malaysia, 11800 Minden, Pulau Pinang, Malaysia
[2]Computer Science and Information Technology Department, Nawroz University, Iraq

***Abstract*—Cross site scripting (XSS) is one of the major threats to the web application security, where the research is still underway for an effective and useful way to analyse the source code of web application and removes this threat. XSS occurs by injecting the malicious scripts into web application and it can lead to significant violations at the site or for the user. Several solutions have been recommended for their detection. However, their results do not appear to be effective enough to resolve the issue. This paper recommended a methodology for the detection of XSS from the PHP web application using genetic algorithm (GA) and static analysis. The methodology enhances the earlier approaches of determining XSS vulnerability in the web application by eliminating the infeasible paths from the control flow graph (CFG). This aids in reducing the false positive rate in the outcomes. The results of the experiments indicated that our methodology is more effectual in detecting XSS vulnerability from the PHP web application compared to the earlier studies, in terms of the false positive rates and the concrete susceptible paths determined by GA Generator.**

Keywords—Web Application Security; Security Vulnerability; Web Testing; Cross Site Scripting; Genetic Algorithm

I. INTRODUCTION

Software systems have been deployed to the public with unexpected security holes. The reason for these security holes is mainly the short time frame of this program's development [1]. Although research on security programs is modern, effective solutions are highly demanded because of the importance of creating programs that are secure and less vulnerable to attacks [2,3].

By injecting malicious scripts into web applications, cross-site scripting (XSS) vulnerabilities are one of the most common security problems in web applications [4,5]. XSS is chosen as the major threat for web application because it provides the surface for other types of attacks, such as session hijacking and Cross Site Request Forgery (CSRF) [6]. XSS can cause damage to both website owners and users. It easily exploits but is difficult to mitigate. Many solutions have been proposed for their detection. However, the problem of XSS vulnerabilities in web applications still persists [7].

To determine XSS vulnerability, the majority of researchers have employed dynamic, static, and hybrid analyses. However, the outcomes achieved by them are marred by the false positive rate and the various challenges in determining XSS vulnerability [8,9]. Consequently, genetic algorithm ventured into the software testing arena by generating test cases for scrutinising the software security. This kind of algorithm offers solutions to determine XSS vulnerability with a lower false positive rate [3,10,11]. Within the Java web application framework, the genetic algorithm locates the entire XSS vulnerability devoid of any false positive rate in the outcomes [3]. Conversely, and post-execution of the algorithm in the PHP web application, it presents several false positive rates. The high false positive results are because the researchers failed to get rid of the infeasible paths which would not perform at all in the CFG.

This paper aims to strengthen the detection approaches of XSS vulnerability in PHP web applications. Section II reviews related research conducted on the problems of XSS. Section III discusses the concept of web application and describes the web application security and vulnerability. Section IV explains the XSS vulnerabilities and continues with the discussion in regards to detection XSS vulnerability in Section V. In Section VI, we describe our proposed approach and the experiments are described in Section VII. Section VIII presents the results for the conducted experiments and detail discussions are explained in Section IX. Finally, ending with conclusion and future works in section X.

II. RELATED WORK

According to the 10 leading vulnerabilities rankings presented by the Open Web Application Security Project (OWASP), the XSS vulnerability can be termed among the top web application vulnerabilities [2,4]. Shar and Tan [9] employed the static analysis methodology on Java web applications. They noted XSS vulnerability with high false positive results. On several occasions, the usage of static analysis offers a high false positive rate. Shar et al. [12] employed the static analysis for addressing the nodes and dynamic analysis for determining the vulnerable nodes. However, the hybrid methodology espoused by them is marred by the false positive rate of the static analysis and the lack of precision in the dynamic analysis results.

Hydara et al. [3] employed the genetic algorithm for generating test cases for the static analysis. The aim was to determine the tangible XSS vulnerability in the Java source code. Their methodology reduced the false positive rate and they could determine the entire actual vulnerable paths within the Java framework.

With regards to the PHP web application, Andrea and Mariano [11] recommended a methodology to locate reflected XSS vulnerability without doing away with it. This methodology was further enhanced by Moataz and Fakhreldin

[10] for determining all three kinds of XSS vulnerabilities. However, the methodology by Andrea and Mariano [11] intends to locate only reflected XSS vulnerability without putting the genetic mutation operator to its best use. On the other hand, the methodology by Moataz and Fakhreldin [10] further enhanced the one offered by Andrea and Mariano [11] by utilising the database of XSS patterns for revealing the probable XSS vulnerabilities: stored, reflected, and DOM-based XSS. However, their experiments were carried out only on stored and reflected XSS vulnerabilities. Furthermore, their methodology has limited scope as certain paths in the CFG do not perform at all; such paths are termed as infeasible.

According to Burhan and Izzat [13], the infeasible path is any path which cannot be implemented at all by the test cases. The infeasible path is triggered because of the dead codes that represent the statements which can never be implemented and reached.

```
1  <?php
2  $example = "test";
3  If ( isset($b))
4  {
5      echo $b; // infeasible traversed this line
6  }
7  ?>
```

Fig. 1. Example of Infeasible Path in PHP

As can be seen in Fig. 1, Line 2 outlines a variable ($b) and initialises a value (“test”). The condition (if) on Line 3 comprises a function (isset) which ascertains whether the variable ($b) is set and is not NULL. Thus, the print statement (echo $b) on Line 5 does not perform at all as the condition return is false; a variable ($b) exists with a value (“test”) which is not NULL. Hence, we term the path (2-3-5) an infeasible one, given the dead codes triggered by the contradicting logic of the condition “if” (isset($b)).

Burhan and Izzat [13] scrutinised the test cases of paths and noted that few of the paths could never be put to test or are seldom tested or visited by a test case. As per Thomas Ball [14], a path is termed as feasible if certain program executions cross that path and the program’s other paths are deemed infeasible; thus, failure is likely in any probable program execution. Typically, infeasible paths generate programs which are quite tough to comprehend. According to Ball, T. and Balakrishnan et al. [14,15], the programmers should reveal paths that are actually executable and those that are not. The outcomes achieved by Moataz and Fakhreldin [10] can be debated, as they detect few of the paths as vulnerable, which they in fact termed as infeasible and would not perform at all.

Although there are several methodologies employed for detecting XSS vulnerability [7,10,11,12,16,17], the threats of XSS continue to persist. Thus, the aim of this paper is to enhance the detection methodologies by eradicating the infeasible paths, thereby reducing the false positive rate of locating XSS vulnerability.

III. Web Application

A web application is a program that executes tasks over a network connection on a web server [18]. Such an application has to be accessed by means of an Internet browser. The web application is used to link the networked tools to the systems. Fig. 2 shows how a user browser and a web server are related.

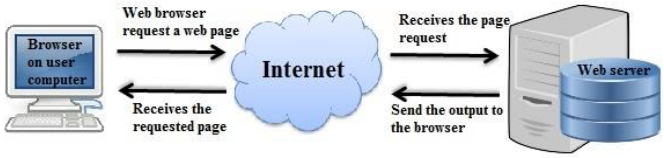

Fig. 2. Relation between User Browser and Web Server

ASP.NET, PHP, and Java server pages (JSP) are few of the well-known technologies which aid software developers in developing dynamically generated web pages [19]. The statistics show that PHP web applications are the most frequently utilised [19,20].

According to Sun et al. [21], securing web applications is imperative today. This security should be fortified with multiple of techniques for bolstering web applications and alleviating attacks. Cross-site scripting is a common vulnerability that enables attackers to insert malicious scripts into the PHP source code. In this case, those web applications are exploited which fail to corroborate the user input.

Thus, this paper emphasises on vulnerabilities pertaining to input validation, considering that input validity is a major web application security vulnerability (SQL injection, cross-site scripting) [5]. Inputs venture into an application from entry points (e.g., $_GET) and take advantage of a vulnerability by connecting to a sensitive sink (for example, mysql_query). The safeguard of the applications can be ensured by consigning sanitisation functions in the paths among the entry points and sensitive sinks. The following section discusses and elucidates in detail the XSS and its vulnerability.

IV. Cross-Site Scripting (XSS)

The vulnerability of web applications is increasing, considering their growing use in day-to-day life. Among the contemporary web applications, XSS is the most exploited security issue [5,21]. Cross-site scripting, as an injecting variant, manipulates the client-side script implemented by the targeted browsers. XSS takes place when a web application utilises an un-encoded or invalidated user input within the output it creates. XSS can trigger major damages for the user or at the site by inserting the malicious scripts into the place where a web application admits user inputs. Inputs that are invalidated can cause transferring of private data, and stealing of cookies and user accounts [2,4]. In other words, the XSS flaw is triggered by un-sanitised or un-validated input parameters. Generally, there are three kinds of XSS attacks – reflected, stored, and DOM-based [6].

Stored XSS strikes when the inserted script is stored in the server (for example, input field or database) [6]. Thus, the browser would be exposed to risk once it retrieves the script from the server. In case of reflected XSS, the malicious script is injected in the website elements (error message). The attacker comes up with a fabricated URL that comprises a malicious script code and entices the targeted user to believe that the URL is genuine [22]. The malicious links are dispatched to the targeted users by email or inserting the link in

a web page which is located on another server. Once the user clicks on the link, the inserted code travels to the attacker's web server, and the attack is then dispatched back to the browser of the victim. Conversely, A document object model (DOM)-based XSS is actioned on the client side. It is initiated by inserting the malicious script in a part of the page's HTML source code [23]. In case of stored and reflected XSS, the targeted users can observe the vulnerability payload in the response page. However, in case of DOM, it can be noted only by scrutinising the page's DOM or on runtime.

The stored and reflected XSS vulnerabilities exploit the client or server sides but the DOM-based XSS exploits only the client side. The researchers are still looking for an effectual means of determining XSS vulnerability in the source code, particularly for stored and reflected as these two are more commonplace compared to DOM-based XSS [6]. The following section outlines the methodologies employed for detecting XSS vulnerability.

V. Detection of XSS Vulnerability

Detecting XSS Vulnerability is the process of addressing and allocating the invalidated inputs or scripts that allow the attacker to inject the malicious script in the source code. The most popular approach to detect vulnerability can be classified into static, dynamic, and hybrid analyses [18]. Static analysis is a method that finds errors in early development that is before the program is initiated [16]. Dynamic analysis detects vulnerabilities by analyzing the information obtained during program execution [24]. The combination of static and dynamic analyses is a hybrid approach; dynamic analysis techniques improve the false alarms of static analysis approaches and provide accurate results [12]. However, experimental results show that a straightforward hybrid approach is unlikely to be superior to a fully static or a fully dynamic detection [8].

Genetic algorithms (GAs) have entered the security field of software testing which is assigned to solve large problems. Gas is a metaheuristic optimization algorithm based on the model of evolution. GAs work as a client application in which the population evolves toward overall fitness even though individuals perish. GAs follow natural evolution mechanisms (e.g., mutation, crossover, and selection), which evaluate the fittest, to solve problems [17]. The elementary genetic algorithm steps are converted into a pseudocode (Fig. 3).

```
population = generate_random_population();
        for(T in vulnerable paths) {
  while(T not covered AND attempt < max_try) {
        selection = select(population);
      offspring = crossover(selection);
      population = mutate(offspring);
          attempt = atempt + 1;
              }
             }|
```

Fig. 3. Genetic Algorithm Pseudocode [3]

A GA begins by initialising an initial populace in a random manner for generating test cases for determining a solution. The fitness function examines whether one of the populace has attained the solution or not. A closer chromosome to the solution indicates a higher fitness value and a higher likelihood of being chosen in next generation. The selection phase selects the closest chromosome for the solution (high fitness value) to execute the mutation and crossover operators so as to generate a new chromosome that possibly can be the solution. A crossover operator generates a new solution by blending two chromosomes, whereas the mutation operator modifies the chromosome values. The fitness function again examines the new chromosomes and whether the solution is attained and is present in one of the new chromosomes.

GA has been observed to be effective in generating solutions for issues related to application software. However, it has not been sufficiently exploited for PHP web security testing. GA was espoused by Andrea et al. and Moataz et al. [10,11]. Notably, the methodology by Andrea and Mariano [11] intends to find out only the reflected XSS vulnerability without utilising the genetic mutation operator to the best of its ability. On the other hand, the methodology by Moataz and Fakhreldin [10] upgraded the one espoused by Andrea and Mariano [11] utilising the database of XSS patterns to reveal the likely XSS vulnerabilities: reflected, stored, and DOM-based XSS. However, their experiments were carried on only stored and reflected XSS vulnerabilities. Furthermore, the results obtained were noted to be imprecise as some paths did not perform at all as per the literature [13,14,15]. Hence, we eliminate the infeasible path from the CFG to attain more favourable results than those from Moataz and Fakhreldin [10], who failed to eliminate paths in PHP web applications.

VI. Proposed Approach

This study improves the confidence in the security of PHP web applications by removing the infeasible path from the CFG to obtain better results compared with those from Moataz and Fakhreldin [10], and generating a test data to uncover XSS vulnerabilities if they exist. The problem lies in generating the minimal number of test cases as an optimization search problem to uncover potential XSS vulnerabilities. Accordingly, a corresponding objective function is used, and it is referred in evolutionary computational techniques as a fitness function.

The detection process starts from Pixy, where it analyzes the PHP script to report on the vulnerable state (Which is to be exploited by an attacker by injecting the XSS script). Based on the outcome produced by Pixy, a Control Flow Graph (CFG) is drawn manually, which reveal the entire vulnerable paths that exist in the PHP script. However, some of these paths may be infeasible in nature, hence would not be executed. Consequently, these paths will be removed, and the GA generator will only be executed on the feasible paths to detect the actual XSS vulnerability and reduce the false positive rate of the present results. The general architecture of the proposed approach is illustrated in Fig. 4.

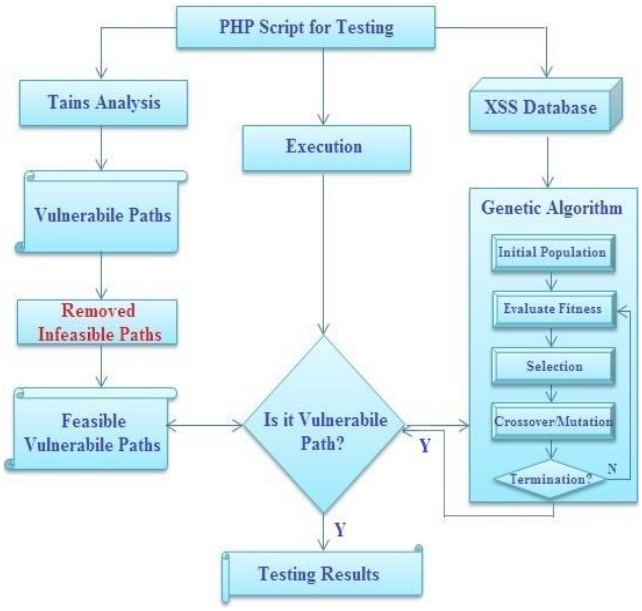

Fig. 4. The General Architecture of the Proposed Approach [10]

In more details, the proposed GA generator actually produced test cases for the feasible vulnerable paths, which subsequently reveal the paths that traverse to the targeted paths. The algorithm begins by initializing a random population (XSS scripts from the database built by the author) as inputs to the PHP script followed by the evaluation of the fitness result of the population. The fitness function evaluated the results of each individual of every generation to understand if these paths are traversing to the targeted paths. A crossover and mutation operator will get a new individual, followed by the proposed GA generator to produce a test case for the new individual, consequently obtaining a new solution and various test case results. In each generation, the fittest individual will be saved and chosen for the next generation.

For the case of paths that are considered vulnerable, if the GA generator returns a zero fitness value for these paths, then the conclusion will be that the input of PHP script (from XSS database) can traverse these paths and execute the sanitized statements. However, if GA generator failed to traverse the targeted paths, then it will be considered as safe, because the proposed GA generator then failed with the XSS input (from XSS database) to traverse these paths.

A. XSS Database

XSS attacks usually injected the malicious scripts in the URL or HTML forms of web applications, which receive PHP functions such as ($_GET or $_POST). The malicious scripts are formed to be executed as application codes, where it can lead to altering the produced content resulting from the injection of a malicious code. Different XSS patterns are collected from various Internet sources [25,26] and stored in a well-organized database to assist GA to generate a test cases to find XSS vulnerable paths.

B. Static Analysis

A tainted variable refers to the inputs from the user or database for XSS vulnerabilities and to print statements that append a string into a web page. Static taint analysis tracks the tainted or untainted status of variables throughout the control flow of the application and determines if a sensitive statement is used without validation [9].

Pixy [27] used as a tool for the taint static analysis. Pixy takes the PHP source code as input. Then a report is created which lists the potential vulnerable lines in the source code, including the paths that contain sanitization statements. Depending of Pixy report, we build a control flow path manually to reach vulnerable sinks and skip sanitization in the source code.

Afterwards, we remove the infeasible paths that do not execute at all. These infeasible paths cannot be considered vulnerable in accordance with the result of Moataz and Fakhreldin [10]. Afterwards, GA defines the security test cases by resorting the feasible paths that create the execution flow traverse target paths.

C. Genetic Algorithm

GA is a search heuristic that mimics the process of natural selection and genetics. It is used as an automatic generator with a specific fitness function and chromosome format, as well as a well-defined crossover and mutation process to generate the offspring of a new population. The following points discuss these operators along with the chromosome and fitness function.

1) Initial population

The most customary kind of encoding or representing chromosomes in genetic algorithms is the binary format. The genetic algorithm population is a suite of likely solutions for a problem. A chromosome is a set of pairs that contains a parameter name and value. For example,

URL: "login.php?firstname=Ahmad&Lastrname=Khalid"

Corresponds to the chromosome:

{(firstname, Ahmad),(lastname, Khalid)}

To simplify, we do not use the first parameter (i.e. name) but instead use only the value that makes our work less complicated and more efficient in comparison.

2) Selection

This stage intends to choose the fittest chromosome to reproduce as per certain selection techniques. Selection techniques ensure that only the best characteristics are transmitted from the current to the next generation. The various methods for selecting individuals include rank, roulette wheel, tournament, and elitist selections [28]. We used the roulette wheel method in which the probability of each individual to be selected is proportional to the fitness value for the individuals, and it is similar to the method used by Andrea et al. and Moataz et al. [10,11]. Afterwards, and based on the probabilities of individuals, two individuals are selected to produce a new solution by crossover and mutation operations. The fitness function evaluates the new offspring and selects the fittest to reproduce for the next generation.

3) Crossover and Mutation

The crossover operation combined two chromosomes to reproduce a new solution with better traits. On the other hand and according to specific mutation probability, the mutation operation occurs by altering the chromosome values.

In this paper, we use a uniform crossover to enable the parent chromosomes to contribute the gene level rather than the segment level. On the other hand, we utilize another method for mutation operation by switching between the attributes values randomly, where the switching will be with the attribute values using XSS scripts from our database. On the basis of the studies by Andrea et al. and Moataz et al. [10,11], we use 0.5 as the best rate for crossover and mutation operations.

4) Fitness Function

Fitness function is aims to evaluate the solution if it is close to the target solution. The best solutions are selected after each generation for the next stage, and genetic operators are used with them. In our work, we choose the fitness function by Moataz and Fakhreldin [10], in which each generation is computed depending on the number of factors that clearly cover each generation. The fitness function of Moataz and Fakhreldin [10] evaluates the script execution path using a specific input. It is composed of several components: the percentage of missing nodes in the path under test, the distance between target and current traversed paths, the importance of the XSS pattern, and the percentage of XSS database coverage.

An individual will cover the vulnerable path if it traverses all of the branches in the path. For example, if a vulnerable path has 10 branches and an input succeeds in traversing all 10 branches, the fitness function will obtain a value of 1, and if the input succeeds in traversing 2 branches, the fitness function will have a value of 0.2, and so on. If the fitness value is greater than the specific threshold, then the individual will survive and will be selected to reproduce for another round. The input distance is equal to zero in case of a string type; if the input type is numeric and not string, then the distance will be calculated as the difference between the traversed and the target paths in term of values using Korel's distance [29].

The GA used the XSS database to build the individual. Therefore, we build an importance factor to reflect the importance of the input used to cover a path. Each pattern previously used in certain files will be saved. Furthermore, we can determine when we can use the same pattern again. The importance will be zero "I = 0" if the input has been used before. The importance will be one "I = 1" if we not used this input before to cover this path. We also examine a case in which we have two inputs for the program. If the value of the first input is used previously as the value for the second input, then the importance will be "I = 0.3".

Another factor in our fitness function reflects the percentage of our XSS database used to cover a path. This factor is used to ascertain that the GA selects different kinds of XSS patterns to cover a path. If we obtain a high percentage, then the GA will be more confident in covering this path and it will exercise it with a different XSS pattern. The database percentage starts from zero when we begin to cover a new path. Evidently, this value is also zero in the initial population. Therefore, our fitness function is [10].

$$F(x) = ((Miss\% + D) * Importance * DB\%) / 100$$

Where F(x) is the fitness value for individual x, Miss% is the missing node percentage in the path using the current individual. D is the distance calculated as the difference between the traversed and target paths, Importance is the importance of the input values, and DB% is the XSS database percentage used to cover the current path.

We attempted to minimize the fitness value so that we can reach a stage in which the current path has no missing node. The path coverage percentage is 100%, and thus we can say the target path is solved completely with the current individual. Furthermore, the current individual successfully forces the PHP script into the target path, and then individual that leads to this outcome as our test data is stored.

VII. Experiments and Analysis

The evaluation is carried out by applying the GA approach, where it is found that a number of paths in the results should be deleted. These paths are infeasible, but considered as vulnerable. Furthermore, the comparison depends on the number of actual vulnerable paths detected by the GA generator. Hydara et al. [3] evaluated the research outcome by depending on the number of actual vulnerable paths detected by the GA generator. Therefore, the aim of the present research is to perform a comparison similar to Hydara et al. [3] within the context of PHP web application.

In this paper, two different experiments are conducted. The first experiment is a Simple Login Script, which contained the reflected XSS vulnerability. The second experiment is a Newspaper Display Script, which contained the Stored XSS vulnerability. We chose these two experiments because our work looking to describe the lacking in Moataz and Fakhreldin [10] approach and minimize the false positive rate in their results, in a way to improve the detection approaches of XSS vulnerability in PHP web application. These two experiments considered different input types, namely either strings and/or numeric. The experiment is conducted by applying the self-developed GA-based test data generator. During the execution of the experiment, the sets of operations are equivalent to the number of feasible potential vulnerable paths that are reported in the static analysis.

A. Simple Login Script Experiment [Reflected XSS]

This experiment contained the Reflected XSS vulnerability, which requested the user to enter his/her first name and last name. Thereafter, the PHP script validated the user inputs to ensure it as a valid input and does not contain XSS patterns or empty strings, which usually occurred in Web forms. Although there are security vulnerabilities in this code, such as the htmlspecialchars, but it's still vulnerable to XSS attacks. Fig. 5 illustrates the HTML form of the experiment, where the user entered the inputs to the PHP script. Fig. 6 illustrates if the code precisely checks the supplied inputs for a string that starts with '<script', which is mandatory for any XSS pattern to execute.

Fig. 5. HTML Form for Simple Login Script

```php
<?php

$a = $_GET["firstname"];  //retrieve the value of First Name input
$b = $_GET["lastname"];  //retrieve the value of Last Name input

  if(substr($a, 0, strlen("<SCRIPT"))=== "<SCRIPT" ) {
    $a=htmlspecialchars($a);   }
   if(isset($b)) {
    $goonb = true;   }
else {
    $goonb = false; }
  if ($goonb ) {
    $b=htmlspecialchars ( $b );   }
    echo $a;   // sensitive s ink
  if ( $goonb ) {
        echo $b;  // sensitive s ink
}
?>
```

Fig. 6. PHP Script of Simple Login Script

Burhan and Izzat [13] defined that the feasible path is any path that can be executed by test cases, and the infeasible path as any path that cannot be executed by test cases. Therefore, the infeasible paths should be removed from the whole paths to effectively to minimize the amount of false positive during the detection process. Fig. 7 depicted both the feasible and infeasible paths in a Simple Login Script experiment.

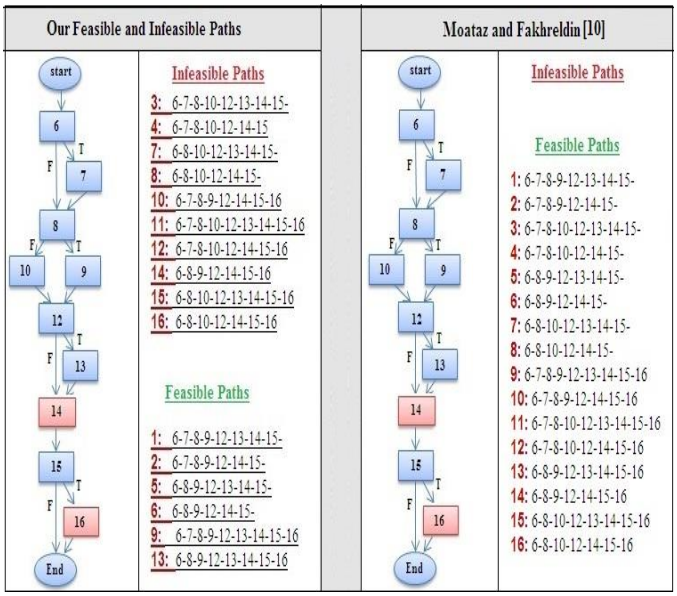

Fig. 7. HTML Form for Simple Login Script

Fig. 7 exhibited the difference between the present study and the study by Moataz and Fakhreldin [10] for both the feasible and infeasible paths, where they considered all the paths as feasible. However, Burhan and Izzat [13] stated that some of the test cases of the paths may never or hardly be tested or visited by any test cases. Ball, T. and Balakrishnan et al. [14,15] reported that the programmers must determine which paths were truly executable and non-executable. As a result, the infeasible paths are removed from the target of the proposed GA generator, with an objective of minimizing the amount of false positive numbers in the obtained results. Fig. 8 described the reasons of each infeasible paths that to be removed.

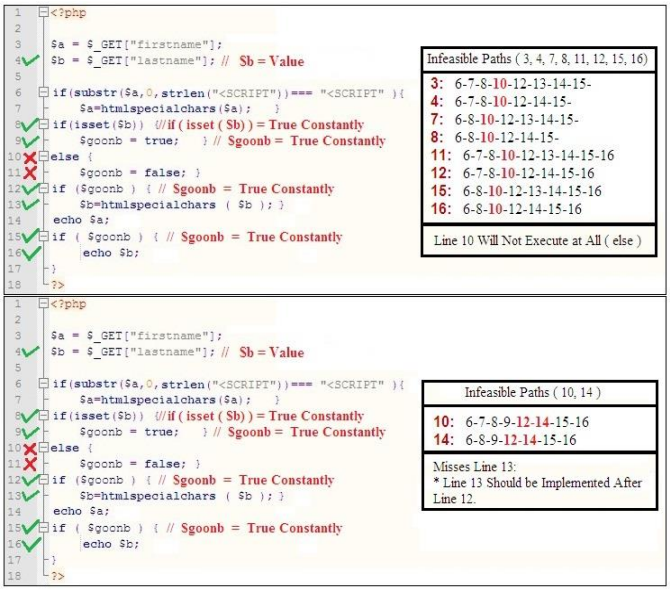

Fig. 8. HTML Form for Simple Login Script

Paths (3, 4, 7, 8, 11, 12, 15 and 16) will be removed because these paths contained the execution of else statement at Line 10, therefore the else statement will not be executed, because the variable $b is defined and assigned as a value. As a result, the Condition (isset ($b)) at line 8 will be TRUE constantly, and else statement will not be executed at any instance.

Paths (10 and 14) will be removed, because these paths contained the execution of the condition at Line 12, hence it must be executed as the htmlspecialchars() statement at Line 13 at every instance.

After the removal process of the infeasible paths from the whole paths, the feasible paths will be the target of the proposed GA to generate test data, which forced the program to flow through these potential vulnerable paths to test on the vulnerability.

However, the proposed GA is unable to read every line of the PHP code, thus the PHP code is required to be probed in an approach to obtain the execution path for any inputs. The PHP code is probed similar to Moataz and Fakhreldin [10], where PHP language constant (__LINE__) is used. This constant (__LINE__) exhibited whether the line of code is executed or not during the program execution.

The probed PHP script is then converted into a PHP function, with an objective of allowing the proposed GA-based test data generator to use the inputs of the function as a parameter to execute the function with the XSS patterns from the XSS database as inputs. Our PHP function will be written as:

Function function_name (Parameter 1 , Parameter 2)

The GA tool copied the probed PHP script and transformed it as one of its own function, which easily executed the function by using XSS patterns as the inputs. The first population is selected randomly from the author's XSS database, followed by GA being executed for many rounds on the test path. After each generation, the proposed fitness function evaluated the solution of the test generator and stores it in each individual. Furthermore, the precisely fitted individuals have the fitness values stored in each rounds. Thereafter, the proposed test generator selected the survivors depending on the fitness value of each individuals by using roulette wheel, where same operator of Moataz and Fakhreldin [10] is used to generate the solutions. The parameter used in the proposed GA generator is presented in Table I.

TABLE I. GENETIC ALGORITHM PARAMETERS FOR SIMPLE LOGIN SCRIPT

Parameter	Values
Population Size	30
Survivor	3
Maximum # Generation	20
# input within one individual	2
Type of inputs	Strings
Crossover rate (Probability)	0.5
Mutation rate (Probability)	0.5

B. *Newspaper Display Script [Stored XSS]*

In this section, the experiment on Stored XSS vulnerability is investigated. The PHP script implemented a simple newspaper display page that allowed users to view topics of specific writers, all the writers in the newspaper, and the articles stored in a MySQL database. If users desire to view an article, the HTML form need to be completed which directly communicates to the server via an URL. The following URL is an example:

http://www.localhost/?name=Ahmad&disply_mode =1

This particular URL contained two values, namely name = Ahmad and disply_mode = 1. However, the implementation of this program can be carried out by posting the written articles' titles or posting the content of the articles of the writers from the MYSQL database. Thereafter, according to the display mode and writer's name from the database, the 'echo' statement at line 21 and 22 will print the writer's name and database's content. However, there are security vulnerabilities in this code including XSS attaches (e.g. htmlspecialchars). Fig. 9 demonstrated the HTML form of the experiment, where the user entered the inputs to the PHP script. Fig. 10 showed that the code precisely checked if the supplied inputs contained a string that starts with '<script', which is mandatory for any XSS pattern that to be executed.

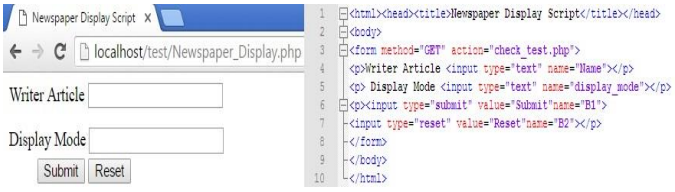

Fig. 9. HTML Form for Newspaper Display Script

```
1  <?php  $Mode = $_GET["disply_mode"] ;  // Display Mode receive Numeric value
2  $Name = $_GET["Name"] ;  // Name of writer recieve Stirng value
3  if ($Mode==1)
4  {
5  $disply_String= select_DBcontent(0);   // Content from Database about the writer
6  }
7  else
8  if ($Mode==2)
9  {
10 $disply_String= select_DBcontent(1); // Content from Database about the writer
11 }
12 else
13 if ($Mode>=3)
14 {
15 $disply_String= "No content for this writer"; // No Content for this writer
16 }
17 if (substr($name, 0, strlen("<script>"))=== "<script>")
18 {
19 $name=htmlspecialchars($name ) ;
20 }
21 echo"The Journalist Name :",$name;  // Sanitize
22 echo $disply_String;  // Sanitize
23 ?>
```

Fig. 10. PHP Script of Newspaper Display Script

As depicted in Fig. 10, the condition of subsutr() function (at line 17) will be true only if strlen() function returned a value of more than zero (true). Therefore, if the variable $name retrieved '<SCRIPT>' value from the first input of the HTML form, then the condition will be true, and followed by executing the sanitization statement of htmlspecialchars() to achieve safety from XSS vulnerability, thus the variable $a is considered safe. However, XSS attack can inject the malicious script with another javascript tag, such as the ("" or "<body background = "javascript:alert('XSS');">"). Hence, the condition (in line 17) failed to cover the malicious script, and the variable $name would not be considered safe.

On the other hand, the variable ($Mode) assigned a numerical value from the second input of the HTML form based on three conditions to assign a value to the variable ($display_String). The first condition is to check if the variable ($Mode) equivalent to 1, then the variable ($display_String) will obtain a value from the database content, where the content can be the XSS script. Thus, the print statement of this variable at line 22 will not be considered safe. The second condition check is if the variable ($Mode) equivalent to 2, then the variable ($display_String) will obtain a value from the database content, which it may contain with the XSS script. Due to the second condition, the print statement process of this variable at line 22 will not be considered safe. The last condition check is if the variable ($Mode) is equivalent to 3, resulting in the variable ($display_String) obtaining a String value. Therefore, the print statement of this variable at line 22 will be considered safe. Fig. 11 shown the three conditions to check the variable Mode of Newspaper Display Script experiment.

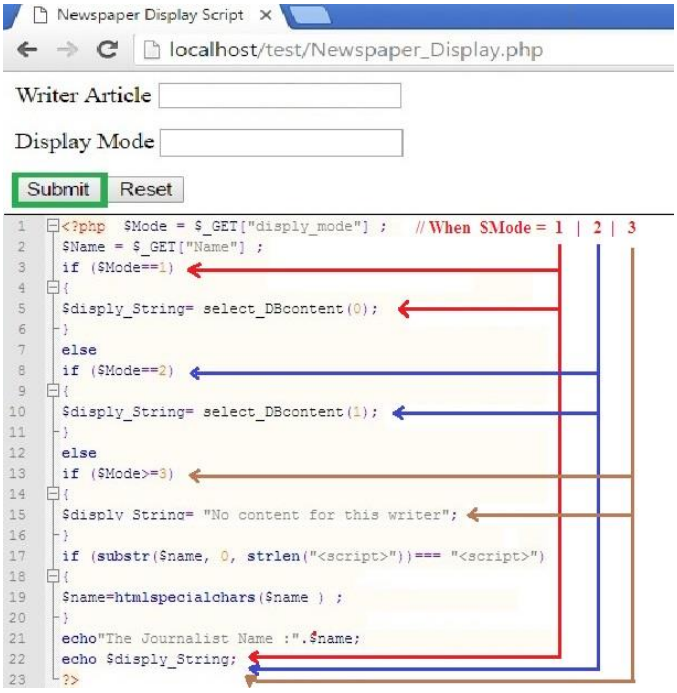

Fig. 11. The Three Condition of ($Mode) to be Executed

When the value of the variable ($Mode) is either 1 or 2, then these paths may contained the XSS scripts from the database, which will then be considered as vulnerable. Furthermore, the three conditions will not be executed alongside, because each of these conditions required different states of condition, such as ($Mode =1, $Mode =2 or $Mode>=3).

Pixy reported the first vulnerability of the experiment, which is the print statement of the variable ($name) at line 21. This particular vulnerability is reflected and may consider as XSS script initiated from the user. The second result of the Pixy is the print statement of the display mode variable at line 22, which can be considered as XSS script from the database due to the lack of validation during the insertion phase. Once the report is completed, the vulnerable path will restart from the line 1 up to the last line (line 22) of the PHP script. Therefore, the PHP script converted the CFG from line 1 to line 22, in an approach that defined the different paths of the program.

The CFG contained 8 infeasible paths that should be removed. In order to define the infeasible path, the understanding of the structure of the script needs to be established. The infeasible path only has a concern towards the print statement of the variable ($Display_String) at line 22. Firstly, the variable ($mode) contained a numerical value of "1". The next condition at line 3 checks if it is equal to 1, followed by returning a value from the database. Therefore, the next condition will not be implemented at line 8, because it checks if the value is equivalent to 2, so that the condition will be FALSE and the statement of the variable ($Display_String) will not be executed at line 10. Similar scenario will be applied for the third condition at line 13, because it checks if the value is equivalent to 3, so that the condition will be FALSE and the statement of the variable ($Display_String) will not be executed at line 15. In total, there are 3 lines that should not be executed alongside, namely lines 5, 10 and 15. In other word, the program should only execute one line from these lines. Furthermore, if the path contained more than one line that is originating from these lines, then it should be removed due to being an infeasible path in nature that will not be executed at all.

The infeasible paths in this context are 1, 2, 3, 4, 5, 6, 9, and 10, where paths 1, 2, 5 and 6 contained two implemented conditions, which are located at lines 5 and 15. Paths 3 and 4 contained two implemented conditions, which are located at lines 5 and 10. The last two infeasible paths 9 and 10 contained two implemented conditions located at lines 10 and 15. Fig. 12 described the feasible and infeasible paths of the Newspaper Display Script experiment which shown the differences between the present study and the previous study by Moataz and Fakhreldin [10].

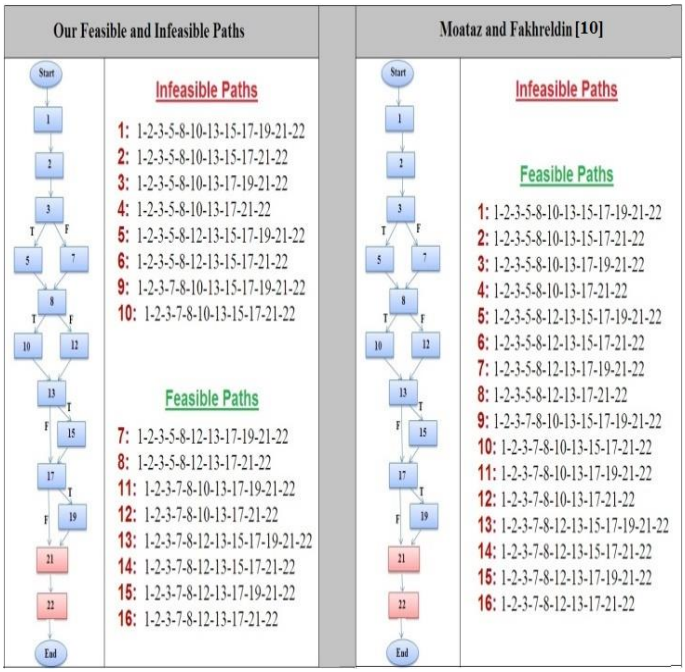

Fig. 12. The Three Condition of ($Mode) to be Executed

Moataz and Fakhreldin considered all the paths as feasible, however the present study removed the infeasible paths from the target of the GA generator. Fig. 13 illustrates the reasons of removal of each infeasible path.

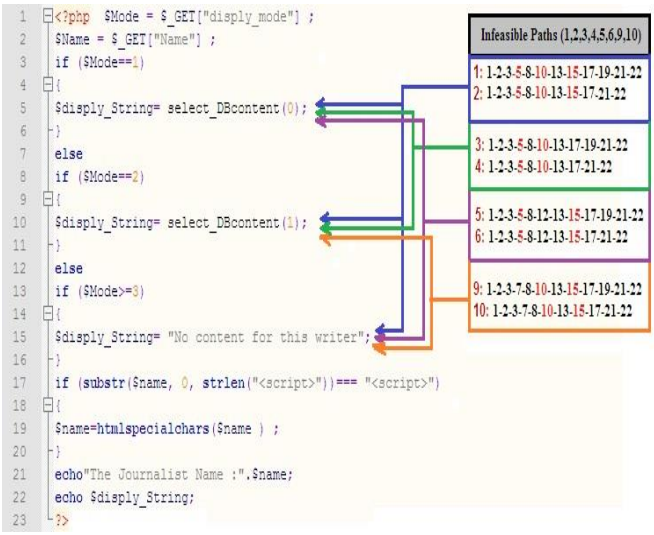

Fig. 13. Describe the Infeasible Paths in Newspaper Display Script

Paths 1, 2, 3, 4, 5, 6, 9, and 10 will be removed according the depiction shown in Fig. 13, because these paths contained the execution of more than one condition (line 5, 10 or 15). However, the executing possibility of this experiment is only to execute one condition in each path, while the other conditions will be False and will not be executed.

The feasible paths will be the target of the self-developed GA to generate the test data that forced the program to flow through these potential vulnerable paths, where the objective is to test whether these paths are indeed vulnerable. The PHP code is probed by the PHP language constant (__LINE__) to allow the GA generator to read the lines of the PHP code. The same operator of Moataz and Fakhreldin [10] is used to generate the solutions. The GA parameters that are applied in this experiment is shown in Table II.

TABLE II. GENETIC ALGORITHM PARAMETERS FOR NEWSPAPER DISPLAY SCRIPT

Parameter	Values
Population Size	30
Survivor	3
Maximum # Generation	20
# input within one individual	2
Type of inputs	Strings and Numeric
Crossover rate (Probability)	0.5
Mutation rate (Probability)	0.5

VIII. RESULTS AND COMPARISON WITH OTHER WORK

This section shows the results details of the proposed test data generator. Firstly, the detection process starts from Pixy where it analyzes the PHP script to report on the vulnerable state (which is to be exploited by an attacker by injecting the XSS script). Based on the outcome produced by Pixy, a Control Flow Graph (CFG) reveals the entire vulnerable paths that exist in the PHP script. However, some of these paths may be infeasible in nature, hence would not be executed. Consequently, these paths will be removed, and the GA generator will only be executed on the feasible paths to detect the actual XSS vulnerability and reduce the false positive rate of the present results.

For the case of paths that are considered vulnerable, if the GA generator returns a zero fitness value for these paths, then the conclusion will be that the input of PHP script (from XSS database) can traverse these paths and execute the sanitized statements. However, if GA generator failed to traverse the targeted paths, then it will be considered as safe, because the proposed GA generator then failed with the XSS input (from XSS database) to traverse these paths.

The results obtained herein on the detection part is evaluated relative to the results of Moataz and Fakhreldin [10], whom improved the approach that were proposed by Andrea and Mariano [11]. The evaluation is carried out by applying the GA approach where it is found that a number of paths in the results should be deleted (i.e. It is because these paths are infeasible and considered as vulnerable). Furthermore, the comparison depends on the number of actual vulnerable paths detected by the GA generator. Hydara et al. [3] evaluated the research outcome by depending on the number of actual vulnerable paths detected by the GA generator. Therefore, the aim of the present research is perform a comparison similar to Hydara et al. [3] within the context of PHP web application.

A. Simple Login Script Experiment [Reflected XSS]

The test generator is operated in the experiment to solve one path and repeated to solve rest of the vulnerable paths. There are 6 feasible paths in the PHP script of a Simple Login Script experiment, where the experiment is repeated once for every each paths (a totally 6 times). The results of the experiment for the detection part are illustrated in Fig. 14, where the X axis represented the rounds or the GA generation, and Y axis represented the best fitness value of the population. When the fitness value is equal to zero, it seemed like the GA generator succeeded in traversing through this path, thus it is considered as a vulnerable path. On the other hand, when fitness value is not equal to zero, then the path is considered as a safe path and the proposed GA generator will fail to traverse this path.

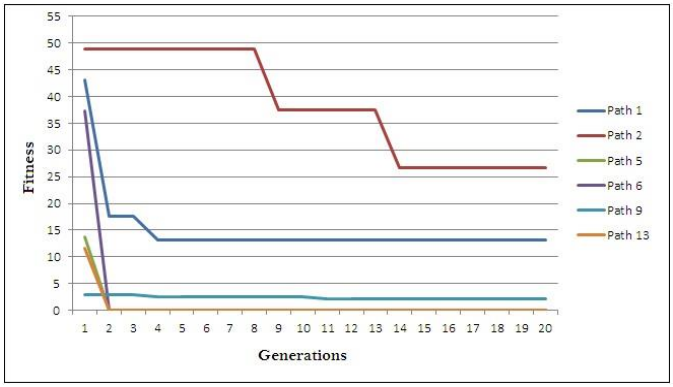

Fig. 14. Detection Results of XSS in Simple Login Script

In Fig. 14, GA is converged for some paths and did not converge for the rest. The paths that the proposed GA approach succeeded to converge are path 5, 6 and 13, with a fitness value of zero from the entire suspected vulnerable paths. The paths 5,

6 and 13 can be considered as vulnerable paths because the three paths skipped the escaping statement (htmlspecialchars). Therefore, it would be a vulnerable paths when the print statement (echo $a) print the variable. We can noted from Fig. 14 that our GA generator choose different scripts for each generation from our XSS database. Once GA generator use any malicious tags except "<script" tag, then the path will traverse the target path and it will get zero fitness values as shown for path 5, 6 and 13.

The comparison of the results of the proposed approach relative to the results of Moataz and Fakhreldin [10] is carried out, where the outcome demonstrate the advancement of the present research in detecting the Reflected XSS. The outcome of the comparison is an improved removal of the infeasible paths, which led to high false positive in the obtained results. Table III presents the results of the research herein and the results of Moataz and Fakhreldin [10] for XSS vulnerabilities detection in a Simple Login Script experiment.

TABLE III. COMPARISON RESULTS OF DETECTION REFLECTED XSS IN SIMPLE LOGIN SCRIPT

Vulnerable Path	Our result	Moataz and Fakhreldin [10]
1: 6-7-8-9-12-13-14-15-	Not Vulnerable	Not Vulnerable
2: 6-7-8-9-12-14-15-	Not Vulnerable	Not Vulnerable
3: 6-7-8-10-12-13-14-15-	Infeasible	Not Vulnerable
4: 6-7-8-10-12-14-15-	Infeasible	Vulnerable
5: 6-8-9-12-13-14-15-	Vulnerable	Not Vulnerable
6: 6-8-9-12-14-15-	Vulnerable	Vulnerable
7: 6-8-10-12-13-14-15-	Infeasible	Vulnerable
8: 6-8-10-12-14-15-	Infeasible	Vulnerable
9: 6-7-8-9-12-13-14-15-16	Not Vulnerable	Not Vulnerable
10: 6-7-8-9-12-14-15-16	Infeasible	Not Vulnerable
11: 6-7-8-10-12-13-14-15-16	Infeasible	Not Vulnerable
12: 6-7-8-10-12-14-15-16	Infeasible	Not Vulnerable
13: 6-8-9-12-13-14-15-16	Vulnerable	Not Vulnerable
14: 6-8-9-12-14-15-16	Infeasible	Vulnerable
15: 6-8-10-12-13-14-15-16	Infeasible	Vulnerable
16: 6-8-10-12-14-15-16	Infeasible	Vulnerable

As shown in Table III, There are some paths considered to be safe paths (i.e. path 1, 2 and 9) and some paths Moataz and Fakhreldin [10] considered it safe which they are infeasible paths and will not execute at all (i.e. path 3, 10, 11 and 12). The false positive rate is the amount of paths that are detected as vulnerable paths, which in actual case are not the actual vulnerable paths. The paths (path 4, 7, 8, 14, 15 and 16) are considered as infeasible paths because the variable ($b) would not be False (at line 10), as shown in Fig. 8. In Line 10, there is else statement, hence considered as infeasible paths and would not be executed (for any inputs or XSS script). One of the special cases is the path 14, where the condition (isset()) is TRUE, but the implementation of the escape function (htmlspecialchars) at line 13 is required, as shown in Fig. 8. As a result, these paths are considered as infeasible and the GA generator would not traverse these paths. Path 5, 6 and 13 are vulnerable paths. However, Moataz and Fakhreldin [10] considered path 5 and 13 as safe paths. Therefore, they detect only one actual vulnerable path which is path 6.

Table IV describes the amount of actual vulnerable paths of this experiment, the amount of the whole paths and the actual vulnerable paths solved (detected) by the self-developed GA generator and by Moataz and Fakhreldin [10] proposed GA generator.

TABLE IV. COMPARISON THE PROPOSED APPROACH RESULTS IN SIMPLE LOGIN SCRIPT

Approach	All Paths Detected by GA Generator	Actual Vulnerable Paths Detected by GA Generator	False Positive
Our GA Generator	3	3	0
Moataz and Fakhreldin [10] GA Generator	7	1	6

The comparison in Table IV exhibited that the self-developed GA performed better compared to the GA designed by Moataz and Fakhreldin [10] in the perspective of the actual vulnerable paths that are detected. The low count in the GA of Moataz and Fakhreldin [10] was due to not removing the infeasible paths from the whole paths.

B. Newspaper Display Script [Stored XSS]

The GA test generator is operated to solve one of the paths and repeat again for the rest of the vulnerable paths. There are 8 feasible paths in the PHP script within this experiment; hence the GA generator is operated once for each paths with a total of 6 runs. The results of the experiment in the detection part are shown in Fig. 15, where the X axis represented the rounds or the GA generation and Y axis represented the best fitness value of the population.

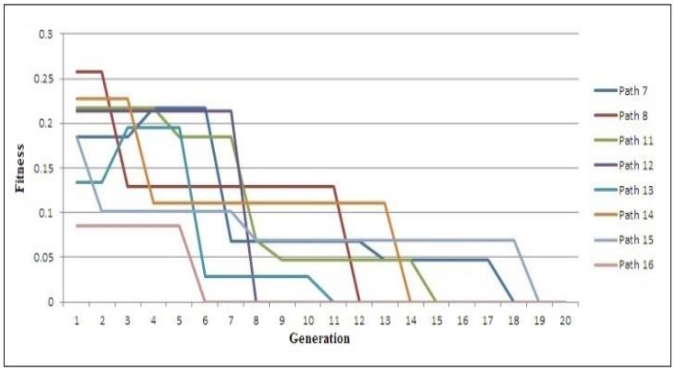

Fig. 15. Detection Results of XSS in Newspaper Display Script

As depicted in Fig. 15, the proposed GA herein succeeded to converge all feasible paths with zero fitness value. The paths 7, 8, 11, 12, 13, 14, 15 and 16 considered as vulnerable paths because our GA generator choose different malicious script in each generation from our XSS database. GA generator choose any malicious scripts from our XSS database and embedded the variables ($display_String and $Name). Therefore, the path would be a vulnerable paths when the print statement (echo

$display_String) print the variable. It is worth to mention that the reason to consider all paths as vulnerable paths because there is no validation (i.e. htmlspecialchars) on the variable ($display_String). Furthermore, these paths are classified as vulnerable because the classification depends on both the input and the sensitive sink that are involved in the path.

Similar to the previous experiment, the proposed approach is compared with the outcome of Moataz and Fakhreldin's [10] approach. The objective of the comparison is to prove that the proposed approach is achieving better than the methodology proposed by Moataz and Fakhreldin [10] for the detection of the stored XSS in the PHP web application. Table V presents the results of the proposed approach and the results of Moataz and Fakhreldin [10] on the detection of Stored XSS vulnerabilities in Newspaper Display Script experiment.

TABLE V. COMPARISON RESULTS OF DETECTION STORED XSS IN NEWSPAPER DISPLAY SCRIPT

Vulnerable Path	Our result	Moataz and Fakhreldin [10]
1: 1-2-3-5-8-10-13-15-17-19-21-22	Infeasible	Not Vulnerable
2: 1-2-3-5-8-10-13-15-17-21-22	Infeasible	Not Vulnerable
3: 1-2-3-5-8-10-13-17-19-21-22	Infeasible	Vulnerable
4: 1-2-3-5-8-10-13-17-21-22	Infeasible	Not Vulnerable
5: 1-2-3-5-8-12-13-15-17-19-21-22	Infeasible	Not Vulnerable
6: 1-2-3-5-8-12-13-15-17-21-22	Infeasible	Not Vulnerable
7: 1-2-3-5-8-12-13-17-19-21-22	Vulnerable	Not Vulnerable
8: 1-2-3-5-8-12-13-17-21-22	Vulnerable	Not Vulnerable
9: 1-2-3-7-8-10-13-15-17-19-21-22	Infeasible	Vulnerable
10: 1-2-3-7-8-10-13-15-17-21-22	Infeasible	Not Vulnerable
11: 1-2-3-7-8-10-13-17-19-21-22	Vulnerable	Vulnerable
12: 1-2-3-7-8-10-13-17-21-22	Vulnerable	Not Vulnerable
13:1-2-3-7-8-12-13-15-17-19-21-22	Vulnerable	Vulnerable
14: 1-2-3-7-8-12-13-15-17-21-22	Vulnerable	Not Vulnerable
15: 1-2-3-7-8-12-13-17-19-21-22	Vulnerable	Vulnerable
16: 1-2-3-7-8-12-13-17-21-22	Vulnerable	Not Vulnerable

Table V shown that the Paths 1, 2, 3, 4, 5, 6, 9 and 10 are infeasible paths, which means these paths would not execute under any circumstances. However, Moataz and Fakhreldin [10] considered these infeasible paths as safe (i.e. path 1, 2, 4, 5, 6 and 10) or vulnerable (i.e. path 3 and 9). However, by operating the present GA generator on these paths, the resulting outcome will be safe, because the GA generator has failed to traverse through these paths.

The proposed GA generator detected 8 actual vulnerable paths, while the GA generator by Moataz and Fakhreldin [10] only detected 3 vulnerable paths from the entire 8 vulnerable paths as shown in Table V. The fundamental reason for the XSS script to traverse these paths and considered the paths as vulnerable is because of both the non-executable nature of the escaping statement (htmlspecialchars) of the variable ($Name) at line 19 (Figure 10) and assignment of a XSS script to the variable ($display_String) at line 3 or 10. Therefore, the print statement (echo $Name) at line 21 or the print statement ($display_String) at line 22 would not be safe, because it may contained the XSS vulnerability.

Moataz and Fakhreldin [10] considered the paths 7, 8, 12, 14 and 16 as safe. However, the escaping statement (htmlspecialchars) at line 19 for the variable ($Name) did not sufficiently secured the path. Thus, the self-developed GA generator has the ability to detect vulnerability paths (6 actual vulnerable paths) with probability high than Moataz and Fakhreldin [10].

Table VI described the amount of actual vulnerable paths occurred in this experiment, the amount of whole paths, and the actual vulnerable paths detected by the self-developed GA generator and the GA generator by Moataz and Fakhreldin [10]. The False positive is the amount of paths detected as vulnerable, which is not the actual vulnerable paths.

TABLE VI. COMPARISON THE PROPOSED APPROACH RESULTS IN NEWSPAPER DISPLAY SCRIPT

Approach	All Paths Detected by GA Generator	Actual Vulnerable Paths Detected by GA Generator	False Positive
Our GA Generator	8	8	0
Moataz and Fakhreldin [10] GA Generator	5	3	2

The results in the Table VI exhibited that the self-developed GA generator performed much better in detecting the actual vulnerable paths compared to the GA generator designed by Moataz and Fakhreldin [10]. Such scenario occurred because Moataz and Fakhreldin [10] did not remove the infeasible paths from the whole paths. As discussed earlier, Moataz and Fakhreldin [10] only detected 5 vulnerability paths, where the 2 paths are considered as infeasible in the present work, which will not be executed in this experiment and considered as false positive results.

IX. DISCUSSIONS

In both experiments, the results shown that the proposed GA generator is better than the GA generator designed by Moataz and Fakhreldin [10], which they presents a high false positive in their results in detection of Stored and Reflected XSS vulnerability. As a conclusion, the result demonstrated the impeccable quality associated with the proposed detection approach, and with this it can be noted that the proposed GA generator performed better than Moataz and Fakhreldin's [10] GA generator in detecting the Reflected and Stored XSS vulnerability within these two experiment for PHP web application. However, more experiment need to be conducted to ensure that the proposed GA generator achieves high accuracy under different experimental environment for Reflected and Stored XSS.

Experiments are conducted herein to detect Reflected and Stored XSS vulnerability within the PHP web application. The results shown that our GA generator detects all actual reflected and stored XSS vulnerabilities in PHP web application without any false positive. On the other hand, Moataz and Fakhreldin [10] detect less actual vulnerable paths with high false positive

in their results, because they did not remove the infeasible paths. The comparison demonstrated that the proposed approach herein enabled the effectively detection of the XSS vulnerability in PHP web application.

X. CONCLUSION

This paper formulated the security testing for XSS vulnerabilities in a search optimization approach, with an objective of eliminating the threat arising from XSS vulnerability in PHP web application. The proposed approach is based on static analysis and genetic algorithm that will be able to detect the XSS vulnerability from PHP source code. Therefore, it was imperative that the present work improved the previous approaches on XSS detection in PHP web application by removing the infeasible paths. The resulting outcome of the present research demonstrated the approach contained zero false positive rates. Furthermore, there was experimentation of detecting the Reflected and Stored XSS vulnerability in the PHP source code, while the approach herein was able to detect the DOM-based XSS attacks based on the self-developed XSS database. However, there were no previous literatures covering experiments on Dom-based XSS. The results demonstrated that the proposed approach achieved better results compared to the previous studies on detection of reflected and stored XSS vulnerability in PHP web applications. It is worth noting here that the proposed approach need to conduct experiments on DOM-based XSS as well, and the proposed approach still need to conduct different experiments on reflected and stored XSS, in a way to reaffirm the proposed approach to detect the XSS vulnerability.

REFERENCES

[1] M.K. Gupta, M.C. Govil, G. Singh, Predicting Cross-Site Scripting (XSS) Security Vulnerabilities in Web Applications', International Joint Conference on Computer Science and Software Engineering (JCSSE), 2015, pp. 162-167.

[2] S. Gupta, B.B. Gupta, Cross-Site Scripting (XSS) attacks and defense mechanisms: classification and state-of-the-art, National Institute of Technology Kurukshetra, Kurukshetra, India, 2015, pp. 1-19.

[3] I. Hydara, A.B.M. Sultan, H. Zulzalil, N. Admodisastro, An Approach for Cross-Site Scripting Detection and Removal Based on Genetic Algorithms', The Ninth International Conference on Software Engineering Advances (ICSEA), 2014, pp. 227–232.

[4] OWASP, top-10 threats for web application security, Available: https://www.owaspp.org/index.php/Top_10_2013, 2013, [Accessed: Feb 2016].

[5] Veracode, State of Software Security, 2014. Available: https://www.veracode.com. [Accessed: April 2016].

[6] V.K. Malviya, S. Saurav, A. Gupta, On Security Issues in Web Applications through Cross Site Scripting (XSS), 20th Asia Pacific Software Engineering Conference (APSEC), 2013, pp. 583-588.

[7] I. Hydara, A.B.M. Sultan, H. Zulzalil, N. Admodisastro, Cross-Site Scripting Detection Based on an Enhanced Genetic Algorithm, *Indian Journal of Science and Technology*, Vol. 8(30), (2015), pp. 1-7.

[8] A. Damodaran, F.D. Troia, C.A. Corrado, T.H. Austin, M. Stamp, A Comparison of Static, Dynamic, and Hybrid Analysis for Malware Detection, J Comput Virol Hack Tech (2015). doi:10.1007/s11416-015-0261-z.

[9] L.K. Shar, H.B.K. Tan, Automated removal of cross site scripting vulnerabilities in web applications, Inf. Softw. Technol., vol. 54, no. 5, 2012, pp. 467–478.

[10] M.A. Ahmed, F. Ali, Multiple path testing for cross site scripting using genetic algorithms", Journal of Systems Architecture, vol. 64, 2015, pp. 50–62. Available from: http://dx.doi.org/10.1016/j.sysarc. 2015.11.001.

[11] A. Avancini, M. Ceccato, Towards security testing with taint analysis and genetic algorithms, in: Proceedings of the 2010 ICSE Workshop on Software Engineering for Secure Systems, Cape Town, South Africa, ACM, 2010, pp. 65-71.

[12] L.K. Shar, H.B.K. Tan, L.C. Briand, Mining SQL injection and cross site scripting vulnerabilities using hybrid program analysis', 35th International Conference on Software Engineering (ICSE '13), 2013, pp 642-651.

[13] B. Barhoush, I. Alsmadi, Infeasible Paths Detection Using Static Analysis, The Research Bulletin of Jordan ACM, Vol. 2, Num. 3, 2013, pp. 120-126.

[14] T. Ball, Paths between Imperative and Functional Programming, ACM SIGPLAN, vol. 34, no. 2, 1999, pp. 21-25.

[15] G. Balakrishnan, S. Sankaranarayanan, F. Ivančić, O. Wei, A. Gupta, SLR: Path-Sensitive Analysis through Infeasible-Path Detection and Syntactic Language Refinement, Alpuente, M., Vidal, G. (eds.) SAS 2008. LNCS, Springer, Heigelberg, vol. 5079, 2008, pp. 238-254.

[16] X. Guo, S. Jin, Y. Zhang, XSS Vulnerability Detection Using Optimized Attack Vector Repertory, International Conference on Cyber-Enabled Distributed Computing and Knowledge (CyberC), 2015, pp. 29-36.

[17] A. Avancini, M. Ceccato, Comparison and integration of genetic algorithms and dynamic symbolic execution for security testing of cross-site scripting vulnerabilities, Information and Software Technology 55, vol. 55, no. 12, 2013, pp. 2209-2222.

[18] M.K. Gupta, M.C. Govil, G. Singh, Static Analysis Approaches to Detect SQL Injection and Cross Site Scripting Vulnerabilities in Web Applications: A Survey, IEEE International Conference on Recent Advances and Innovations in Engineering (ICRAIE-2014), 2014, pp. 1-5.

[19] A. Mishra, Critical Comparison Of PHP And ASP.NET For Web Development - ASP.NET & PHP, International Journal of Scientific & Technology Research, vol. 3, no. 7, 2014, pp 331-333.

[20] CWE, CWE - CWE-79: Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting') (2.5), The MITRE Corporation. Available: http://cwe.mitre.org/data/definitions/79.html. [Accessed: Feb, 2016].

[21] S. Rafique, M. Humayun, B. Hamid, A. Abbas, M. Akhtar, K. Iqbal, Web application security vulnerabilities detection approaches: A systematic mapping study", Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD), 2015, pp. 1-6, doi:10.1109/SNPD.2015.7176244.

[22] G. Dong, Y. Zhang, X. Wang, P. Wang, L. Liu, Detecting Cross Site Scripting Vulnerabilities Introduced by HTML5, International Joint Conference on Computer Science and Software Engineering (JCSSE), 2014, pp. 319-323.

[23] V.K. Malviya, S. Saurav, A. Gupta, On Security Issues in Web Applications through Cross Site Scripting (XSS), 20th Asia-Pacific Software Engineering Conference (APSEC), 2013, pp 583-588.

[24] T.R. Toma, Md.S. Islam, An Efficient Mechanism of Generating Call Graph for JavaScript using Dynamic Analysis in Web Application, International Conference on Informatics, Electronics & Vision (ICIEV), 2014, pp. 1-6.

[25] OWASP, XSS Filter Evasion Cheat Sheet, 2016. Available: https://www.owasp.org/index.php/Testing_for_Stored_Cross_site_scripting. [Accessed: April 2016].

[26] RSnake, XSS cheatsheet. Available: http://n0p.net/php_app_sec/xss.html. [April: May 2016].

[27] Pixy, Pixy: XSS and SQLI Scanner for PHP Programs, 2007. Available: http://pixybox.seclab.tuwien.ac.at. [Accessed: Feb 2016].

[28] [29] M.A. Ahmed, I. Hermadi, GA-based multiple paths test data generator, J. Comput. Oper. Res. (COR) Focus Issue Search-Based Softw. Eng. (SBSE), 2008, pp. 3107–3124. DOI link: http://dx.doi.org/, doi:10.1016/j.cor.2007.01.012.

[29] I. Hermadi, M.A. Ahmad, Genetic Algorithm based Test Data Generator, The 2003 Congress on Evolutionary Computation (CEC '03), 2003, pp. 85-91.